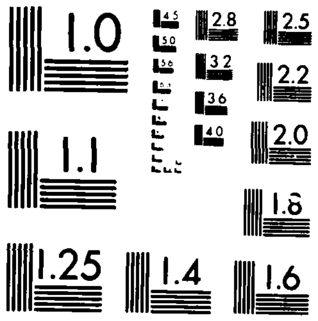

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

(2)

NRL Report 8940

Executing Trace Specifications Using Prolog

JOHN D. MCLEAN, DAVID M. WEISS, AND CARL E. LANDWEHR

Computer Science and Systems Branch
Information Technology Division

January 21, 1986

NAVAL RESEARCH LABORATORY
Washington, D.C.

ADA 164877

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Report 8940	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Research Laboratory; 6b OFFICE SYMBOL (If applicable) 7593	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and ZIP Code) Washington, DC 20375-5000	7b ADDRESS (City, State, and ZIP Code)
8a NAME OF FUNDING / SPONSORING ORGANIZATION Office of Naval Research; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code) Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
61153N	75-2063-0-5	RR014-09-42	DN480-540

11 TITLE (Include Security Classification)
Executing Trace Specifications Using Prolog

12 PERSONAL AUTHOR(S)
McLean, John D., Weiss, David M., and Landwehr, Carl E.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Interim	FROM 1/84 TO 6/85	1986 January 21	14

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Automatic implementation; Rapid prototyping; Formal specification; Prolog; Logic programming

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Trace specifications have some desirable properties that most specifications lack: they are abstract, so they need not impose unnecessary constraints on the implementor; and they are formal, so they can be understood unambiguously and can be rigorously tested for consistency, totalness, and correctness of implementation. Nevertheless, understanding trace specifications and translating them to computer programs are significant tasks. This report documents experiments in translating trace specifications to Prolog so that they can be executed directly. The selection of Prolog over other possible languages is discussed, and several example specifications and their translations are presented. Some generally useful Prolog predicates are gleaned from these examples, and difficulties encountered in performing the translations are described. On the way, an implementation-free semantics for a subset of Prolog is given. The report concludes with a discussion of possibilities for mechanically translating trace specifications to Prolog. Keywords: Logic programming. ←

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT] ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL John D. McLean	22b TELEPHONE (Include Area Code) (202) 767-3852; 22c OFFICE SYMBOL 7593

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

CONTENTS

INTRODUCTION 1

TRACE SPECIFICATIONS 1

STRUCTURE OF A SYSTEM FOR EXECUTING TRACE SPECIFICATIONS 2

PROLOG 3

PROLOG IMPLEMENTATIONS OF TRACE SPECIFICATIONS 6

CONCLUSIONS 9

REFERENCES 9

EXECUTING TRACE SPECIFICATIONS USING PROLOG

INTRODUCTION

The trace method of software specification [1] has a formal foundation that supports unambiguous specifications susceptible to rigorous proof techniques. The method is also abstract and produces specifications that neither make unwanted design decisions nor force the programmer to glean essential program features from a mass of extraneous clutter. Its formal foundation renders traces superior to English-like specifications, and its abstractness renders them superior to procedural specification languages.

A disadvantage to formal, abstract specifications has been their opacity to programmers who use procedural languages. To aid the programmer, we think software tools are needed to support program development. An important tool is a rapid prototyping system that enables programmers to check what is required by a given specification and users to check that what is specified is what they really want. This report describes the approach we are taking to build a system that translates trace specifications into programs. First, we provide an introduction to trace specifications. Next, we describe a general structure for a system for executing such specifications and discuss several alternative approaches for building such a structure. We then give reasons for using Prolog and present some example trace specifications with their translations into Prolog. Finally, we develop an implementation-free semantics for a subset of Prolog that helps us address the possibility of mechanically translating trace specifications into Prolog.

TRACE SPECIFICATIONS

A trace specification for a module consists of two parts: (1) a syntax section that gives the procedure names and types the module comprises, and (2) a semantics section that gives the behavior that the module's procedures must exhibit. Procedure behavior is given by listing assertions that describe the behavior of sequences of procedure calls, written $call_1 \ldots . call_n$, known as *traces*. The assertions are based on first-order logic, supplemented by the predicate L that, when applied to *legal* traces (traces that do not cause an error), is true, and the function V that when applied to a legal trace ending in a function call, gives a return value for that trace. The null trace is always legal. Two traces S and R are *equivalent*, written $S \equiv R$, when they are indistinguishable as far as L and V are concerned with respect to future program behavior. More formally, $S \equiv R$ if and only if for any trace $T, L(S.T)$ *iff* $L(R.T)$ and for nonnull $T, V(S.T) = V(R.T)$, if defined.

As an example, consider the following specification for a stack of integers:

STACK SPECIFICATION

Syntax:
  PUSH: integer
  POP:
  TOP: --> integer
  DEPTH: --> integer

Manuscript approved August 16, 1985.

Semantics:

(1) $L(T) \rightarrow L(T.PUSH(n))$
(2) $T.DEPTH \equiv T$
(3) $T.PUSH(n).TOP \equiv T.PUSH(n)$
(4) $T.PUSH\ (n).POP \equiv T$
(5) $L(T) \rightarrow V(T.PUSH(n).TOP) = n$
(6) $V(DEPTH) = 0$
(7) $L(T) \rightarrow V(T.PUSH(n).DEPTH) = 1 + V(T.DEPTH)$

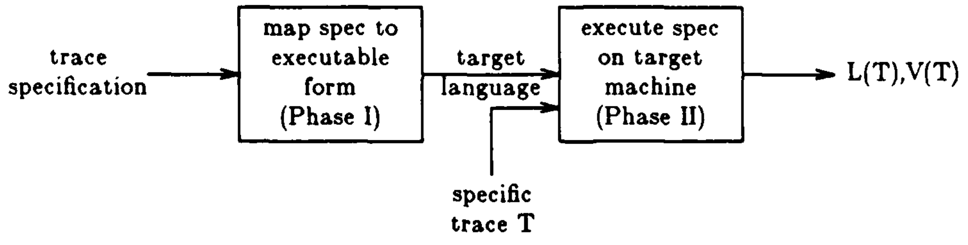
The syntax section says that the stack module consists of the procedures PUSH, POP, TOP, and DEPTH. PUSH takes an integer as a parameter, and TOP and DEPTH return integers. The first assertion of the semantics section says that if a trace is legal, then one can legally append a call to PUSH to the trace, i.e., one can legally push any integer. Assertions (2) to (4) say that certain procedure calls do not alter future program behavior with respect to a trace. In particular, assertion (4) says that a PUSH immediately followed by a POP has no effect on the state of the module. The fifth assertion, when accompanied with previous assertions, says that TOP returns the last item pushed on the stack that has not been popped. Assertions (6) to (7) say that DEPTH returns the depth of the stack. The reader interested in a more detailed exposition of traces and their formal foundation is directed to Ref. [1].

We can derive the legality and value of a specific trace, say PUSH(5).TOP, from the specification as follows:

(i) L(PUSH(5)) [by assertion 1 and the legality of the null trace]
(ii) PUSH(5).TOP $\equiv$ PUSH(5) [by assertion 3]
(iii) L(PUSH(5).TOP) [from lines (i) and (ii) and definition of equivalence]
(iv) V(PUSH(5).TOP) = 5 [by assertion 5 and legality of the null trace]

STRUCTURE OF A SYSTEM FOR EXECUTING TRACE SPECIFICATIONS

Our goal is to develop a system that will accept a trace specification and a specific trace (or traces) *T* as input. As output, it will provide *L*(*T*) and *V*(*T*) as defined by the specification. Graphically it might look like this:


trace specification
map spec to executable form (Phase I)
target language
execute spec on target machine (Phase II)
L(T),V(T)
specific trace T

The choice of the target language is significant. If a conventional, procedural language (*e.g.*, Fortran, Pascal) were chosen, the Phase 1 task would require translating the nonprocedural trace specification into a procedural form—a substantial undertaking not certain of success. If, on the other hand, a target language can be chosen in which the expression of a trace specification is relatively straightforward, but which still admits of execution in Phase II, the Phase I task is greatly reduced. Languages such as Prolog, LISP, SNOBOL, and their relatives, and the input languages for YACC and the Boyer-Moore Theorem Prover fall in this category. In this section we consider briefly the effect of choosing each of these alternatives, and explain why we have proceeded with initial experiments in Prolog.

Approaches considered include:

YACC. YACC is a Unix-based compiler-compiler [2] that accepts grammar rules in BNF and semantic actions in *C* as input, and generates a compiler for the specified language (also in *C*) as output. An approach that uses YACC would call for construction of a YACC program that could accept a trace specification and generate a compiler for that specification. The compiler would execute in Phase II to parse incoming traces and provide values of *L* and *V* as output. A YACC program to translate traces into Prolog has been written. Some difficulties were encountered in constructing a grammar for input to YACC that would guarantee recognition and reduction of traces, but investigation of this alternative is continuing.

Boyer-Moore Theorem Prover. This is probably the most capable automated theorem prover currently available [3]. If trace specifications could be translated into axioms for the theorem prover, the prover could be used to check the value and legality of a given trace. The Phase I tool would have to handle the translation of trace specification into the proper input notation for the theorem prover. Discussions with Moore about the learning time needed to become facile in the use of the prover (approximately six months for a logician) led us to discard this approach.

LISP. Trace specifications could be translated into LISP [4] by Phase I and then executed in Phase II. Although pure LISP can be considered a nonprocedural language, it is well known that programming in pure LISP is impractical. Consequently, this approach would be likely to encounter the same kinds of difficulties as translating trace specifications into a procedural language. However, we are investigating this approach.

Prolog. In this case, trace specifications must be translated into Prolog [5] notation by Phase I and then executed in Phase II. The syntax of Prolog is well-suited for this purpose, and Prolog includes a theorem prover that can be used both to "execute" the specifications and to attempt to prove theorems about them. Prolog is the target language in our chosen approach; more details are provided in the next section. A drawback is that, despite its nonprocedural look, Prolog behavior does depend on the order in which facts are presented (because of its use of a depth-first search).

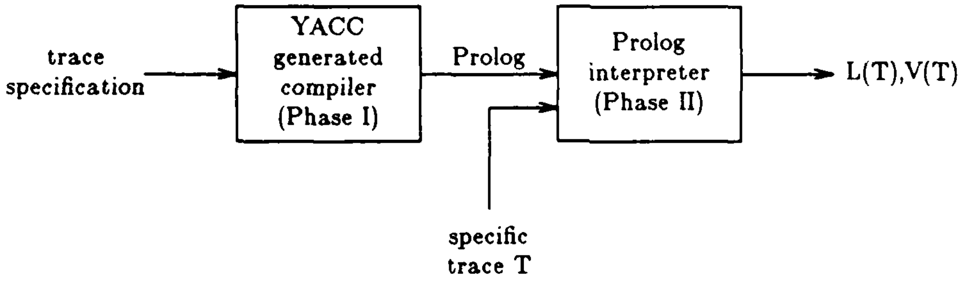
Prolog Variants. Several variants of Prolog exist that may be as well-suited as Prolog for expressing trace specifications and that may solve the problems caused by the sensitivity of Prolog to fact-ordering. Two examples are LOGLISP [6] and TABLOG [7]. We have yet to evaluate either of these fully.

SNOBOL or Icon. Icon [8] is a string processing language descended from SNOBOL [9]. An approach based on either SNOBOL or Icon might resemble that based on YACC: an Icon program would be used to translate a trace specification into an Icon program; this latter program would then be executed with specific traces as input. An Icon program to translate traces into Prolog was written; translation of trace specifications into Icon has not been attempted as yet.

PROLOG

Prolog is a programming language based on Horn clauses [10]. A Horn clause is a formula of first-order logic that is of the form (1) $P(t_1,t_2,\ldots,t_n)$ where P is an n-place predicate and each t_i is a term, of the form (2) $\neg P(t_1,t_2,\ldots,t_n)$ where P and each t_i are as above, or of the form (3) $C \vee \neg P(t_1,t_2,\ldots,t_n)$ where C is a Horn clause and P and each t_i are as above. Terms can either be constants, variables, or functions of other terms. All variables are assumed to be universally quantified. A Prolog program is a set of clauses where each clause is of the first or the third form. Clauses of the first form are called *facts* and written as they are in first-order logic with the understanding that predicate letters, constants, and function symbols begin with small letters, and variables begin with capitals. Clauses of the third form are called *rules* and written as $F_1{:-}\ F_2,\ldots,F_n$ where F_1 is the single nonnegated disjunct of the clause. F_1 is called the *head* of the rule, and $F_2\ \ldots,F_n$ is the *tail*. A clause that is either a *fact* or a *rule* is called a *program clause*.

The translation from trace specifications to Prolog appears to be simple enough that we may be able to use YACC to generate a Phase I program that accepts trace specification language as input and generates a Prolog program as output for execution by the Prolog interpreter in Phase II, as shown below.


trace specification
YACC generated compiler (Phase I)
Prolog
Prolog interpreter (Phase II)
L(T),V(T)
specific trace T

The stack specification translates simply into the Prolog program shown below. In Prolog, traces are considered as lists and written in reverse notation. Hence, *PUSH(n).TOP* becomes [*top*,*push*(*n*)]. The empty sequence of procedure calls is denoted by [ ]. The legality predicate is written *leg*, and *equiv* and *val* denote ≡ , and the relational form of *V* , respectively. Line numbers are included for future reference.

STACK PROGRAM

```
(1) leg([ ]).

(2) leg([push(S)|T]) :- leg (T).
(3) equiv([depth|T],T).
(4) equiv([top,push(S)|T],[push(S)|T]).
(5) equiv([pop,push(S)|T],T).
(6) val([top,push(S)|T],S) :- leg (T).
(7) val([depth],0).
(8) val([depth,push(S)|T],V) :- val([depth|T],R), V is R + 1, leg(T).

(9) leg(T) :- equiv(T,S), leg(S).
(10) val([T|S],V) :- equiv(S,R), val([T|R],V).
(11) equiv(X,Y) :- equiv(A,B), append(R,A,X), append(R,B,Y).
(12) equiv(S,T) :- equiv(T,S).
(13) equiv(S,T) :- equiv(S,A), equiv(A,T).
(14) equiv(S,S).

(15) append([ ],L,L).
(16) append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Clauses (2) to (8) are Prolog translations of semantic assertions (1) to (7) of the trace specification. The remaining clauses are common to all programs produced by our system. Clause (1) and clauses (9) to (14) are translations of assertions from the trace deductive system as described in Ref. [1].

Clauses (15) to (16) define the *append* predicate used to manipulate the list representations of traces. The reason for the particular order of clauses is discussed in the next section.

To illustrate the nonvariance of the common clauses, we include a translation of a queue specification.

QUEUE SPECIFICATION

```
Syntax:
 ADD: integer
 REMOVE:
 Front: --> integer

Semantics:
 (1) L(T) → L(T.ADD(n))
 (2) ADD(n).REMOVE ≡ [ ]
 (3) T.ADD(n).FRONT ≡ T.ADD(n)
 (4) T.ADD(n).ADD(m).REMOVE ≡ T.ADD(n).REMOVE.ADD(m)
 (5) V(ADD(n).FRONT) = n
 (6) L(T) → V(T.ADD(n).ADD(m).FRONT) = V(T.ADD(n).FRONT)
```

QUEUE PROGRAM

```
(1) leg([ ]).

(2) leg([add(S)|T]) :- leg (T).
(3) equiv([remove, add(S)], [ ] ).
(4) equiv([front, add(S)|T],[add(S)|T]).
(5)equiv([remove,add(S),add(R)|T],[add(S),remove,add(R)|T]).
(6) val([front,add(S)],S).
(7) val([front,add(S),add(R)|T],X) :- val([front,add(R)|T],X), leg(T).

(8) leg(T) :- equiv(T,S), leg(S).
(9) val([T|S],V) :- equiv(S,R), val([T|R],V)
(10) equiv(X,Y) :- equiv(A,B), append(R,A,X), append(R,B,Y).
(11) equiv(S,T) :-(T,S).
(12) equiv(S,T) :- equiv(S,A), equiv(A,T).
(13) equiv(S,S).

(14) append([ ],L,L).
(15) append([X|L1],L2.[X|L3]) :- append(L1,L2,L3).
```

Given such a program, Prolog tries to satisfy assertions when prompted. If the assertion contains no variables and can be derived from the program, **yes** is returned. If the assertion contains variables and there are constants that when substituted for the variables of the assertion make the assertion derivable, these constants are returned. Otherwise, Prolog will return **no** or attempt an infinite search for a derivation. When it does one rather than the other is examined in the next section.

Queries submitted to the stack program are illustrated in the following script of a Prolog session. Characters typed by the user are shown in boldface. The version of Prolog used is C-Prolog on a VAX operating under the BSD 4.2 Unix operating system. The stack program is contained in a file named **stack**.

PROLOG SESSION WITH STACK

```
CProlog version 1.2a, NRL-CSS
| ?- [stack].
stack consulted 1304 bytes 0.333333 sec.

yes
| ?-leg([push(5)]).

yes
| ?-equiv([top,push(5)],[push(5)],[push(5)]).

yes
| ?-leg([top,push(5)]).

yes
| ?- val([top,push(5)],5).

yes
| ?- val([top,push(5)],X).

X = 5

yes
| ?- halt.

[Prolog execution halted]
```

PROLOG IMPLEMENTATIONS OF TRACE SPECIFICATIONS

In our discussion of Prolog so far, we have glossed over several issues that are relevant to regarding Prolog programs as implementations of trace specifications. First, we have not said what trace assertions can be cast as Prolog program clauses. Second, we have not said when a clause is derivable from a Prolog program. We address these issues in this section.

It is well known that any formula of first-order logic (and hence, any trace assertion) is equivalent to a formula $(Q1) \ldots (Qn)F$, called its *prenex disjunctive form*, where each (Qi) is a quantifier and F is a disjunction of either atomic formulae or their negations. Such a formula is, in fact, a Prolog program clause if (1) no (Qi) is an existential quantifier, and (2) F contains at least one nonnegated disjunct and at most one negated disjunct. Condition (1) is not problematic since a formula containing an existential quantifier is cosatisfiable with a formula, called a *Skolem formula*, that contains only universal quantifiers with some additional function symbols [11]. Hence, eliminating existential quantifiers from a prenex formula does not alter what we can derive from that formula. Condition (2), however, presents more serious troubles. A formula whose prenex disjunctive form contains too few nonnegated disjuncts or too many negated disjuncts can be converted to a program clause by using the Prolog predicate **not** which has the property that **not(F)** is Prolog-derivable if and only if an attempt to derive F in Prolog leads to a return of **no**. Hence, we can write $\neg G(t)$ as $not\ (G(t))$ and $F(t) \vee G(t)$ as $F(t){:-}not(H(t))$ if we can define a predicate H such that $G(t)$ is Prolog-derivable if and only if $H(t)$ is refutable. However, converting a prenex formula to a program clause in this way often results in a loss of derivational power. To understand this limitation on using **not** we must consider what it means for something to be Prolog-derivable, i.e., we must give a semantics for Prolog.[1]

Call S a *substitution* if it is an assignment of constants to variables. We denote the result of applying S to a formula T by T/S. Given a predicate F and a constant c, $F(c)$ is *refutable* by a Prolog program P if (1) there are no occurrences of $F(c)$ or $F(X)$ in P for any variable X, (2) any rule of the form $F(c){:-}T_1, \ldots, T_n$ in P is such that each fact in the set $\{T_1/S, \ldots, T_n/S\}$ is refutable by P for any assignment S, and (3) any rule of the form $F(X){:-}T_1, \ldots, T_n$ in P is such that each fact in the set $\{T_1/S, \ldots, T_n/S\}$ is refutable by P for any assignment S that assigns c to X. We say that $F(c)$ is *derivable* from P if (1) there appears in P a program statement of the form

$F(c)$,

$F(X)$,

$F(c){:-}T_1, \ldots, T_n$ where each fact in the set $\{T_1/S, \ldots, T_n/S\}$ is derivable from P for some assignment S, or

$F(X){:-}T_1, \ldots, T_n$ where each fact in the set $\{T_1/S, \ldots, T_n/S\}$ is derivable from P for some assignment S that assigns c to X;

and (2) $F(c)$ is refutable by that part of P that precedes that fact cited in (1). Given a query of the form $F(c)$, Prolog will answer **yes** if $F(c)$ is derivable from P, will answer **no** if $F(c)$ is refutable by P, and will fail to answer otherwise.[2]

From this discussion we can see that although any trace specification may have a sound Prolog implementation, i.e., an implementation that does not return incorrect answers to queries, not every sound implementation of a specification is complete. Many implementations may not return an answer to a query even though there is an answer that can be logically derived from the specification.

One source of incompleteness is obviously the use of **not** since we may not be able to refute the assertion being negated; however, this is not the only source. An examination of our initial stack program shows that claims of the form **equiv(trace1,trace2)** are not refutable. This is because of the presence of axioms (11) to (14) whose heads will always be satisfied by the query. Since axiom (9) for legality and axiom (10) for values appeal to the notion of equivalence, legality assertions and value assertions are also not refutable.

On a more positive note, incomplete programs can often be made complete. For example, we can solve the problem with our stack program by dropping the troublesome axioms and adjusting the rest of the program to compensate. Note that axioms (12) to (14) serve only to ensure that equivalence is an equivalence relation. This is necessary to be able to determine correctly whether two traces are equivalent *simpliciter*, but it is not necessary for determining legality or values. For example, we can derive **leg([depth,depth,push(1)])** from **leg([push(1)])** by using (3) to derive **equiv ([depth,depth push(1)],[depth,push(1)])** and **equiv ([depth,push(1)],[push(1)])** without having to appeal to the fact

1. We ignore the Prolog predicate *cut* [5] since although it is used to improve efficiency of Prolog derivations, it does not increase the set of derivable facts. The following discussion can be extended to include *cut* by considering the order of facts within a rule's tail as well as the order of clauses within a program.

2. The response to a query of the form *F(X)* can be determined by considering queries of the form *F(c)* where *c* is a member of the Herbrand Universe [10] of *P*.

that **equiv([depth,depth,push(1)],[push(1)])**. Hence, we can drop axioms (12) to (14) without affecting the program's behavior with respect to legality or values. Axiom (11) serves a different purpose. It allows us to look for equivalent traces within traces and replace them. However, we can eliminate this axiom by redefining equivalence in our specification. Hence, consider the following stack specification:

STACK SPECIFICATION

Syntax:
PUSH: integer
POP:
TOP:--> integer
DEPTH:--> integer

Semantics:
(1) $L(T) \rightarrow L(T.\ PUSH(n))$
(2) $T.DEPTH.S \equiv T.S$
(3) $T.PUSH(n).TOP.S \equiv T.PUSH(n).S$
(4) $T.PUSH(n).POP.S \equiv T.S$
(5) $L(T) \rightarrow V(T.PUSH(n).TOP) = n$
(6) $V(DEPTH) = 0$
(7) $L(T) \rightarrow V(T.PUSH(n).DEPTH) = 1 + V(T.DEPTH)$

We have modified axioms (2) to (4) to make explicit the fact that equivalent traces can be substituted within a trace. From the viewpoint of logical derivation this modification changes nothing, but from the view of Prolog, it enables us to drop a troublesome axiom from our program. Hence, we have the following implementation:

STACK PROGRAM

```
(1) leg ([ ]).

(2) leg([push(S)|T]) :- leg(T).
(3) val([top,push(S)T],S) :- leg(T).
(4) val([depth],0).
(5) val([depth,push(S)|T],V) :- val([depth|T],R), V is R + 1, leg(T).
(6) equiv(A,B) :- append([depth],L1,T), append(H,T,A), append (H,L1,B).
(7) equiv(A,B) :- append([top,push(N)],L1,T), append(H,T,A),
            append([push(N)],L1,L2), append(H,L2,B).

(8) equiv(A,B) :- append([pop,push(S)],L1,T), append (H,T,A), append(H,L1,B).
(9) leg(T) :. equiv(T,S),leg(S).
(10) val([T|S],V) :- equiv(S,R), val([T|R],V).

(11) append([ ],L,L).
(12) append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

This implementation derives all the legality and value assertions that the other derives, but it also refutes assertions involving false legality and value claims. It is inferior to the first implementation in that we can no longer derive **equiv(a,b)** from **equiv(a,c)** and **equiv(c,b)**, but equivalence is important primarily in the role it plays in specifying legality and values. Further, we can define a notion of transitive equivalence if we like by adding the following axioms:

```
(13) tequiv(A,A).
(14) tequiv(A,B) :- equiv(A,B).
(15) tequiv(A,B) :- tequiv(A,C), tequiv(C,B).
```

Summarizing, our semantics has enabled us to determine that a program is incomplete and isolate the cause of the incompleteness. By rewriting the program, we have rendered it complete, but at the cost of not being able to derive any equivalence assertions that depend on the transitive closure of *equivalence*. This does not bother us since we are rarely interested in equivalence assertions *per se*, but only in so far as they contribute to the derivation of legality and value assertions.

CONCLUSIONS

We conclude that Prolog warrants further research as a target language for a trace implementation system. It is possible to translate a trace specification into a sound Prolog implementation although care must be taken in formulating the specification if we are to avoid incomplete programs. Further, translation can probably be done mechanically although it is not clear that there is a mechanical procedure that will always yield a complete program. Nevertheless, we have been able to isolate a major cause of incompleteness in translated programs and have shown a method for writing specifications that eliminates this cause. Work must proceed to see how far this method can be generalized. We are interested in whether all specifications can be formulated to yield complete programs and if not, whether we can describe those that cannot be so formulated.

In the meantime we are also working on other approaches. We have had some success with LISP. It is possible that the two approaches may complement each other even if neither is completely successful. In this case, a combination of Prolog and LISP, such as is found in LOGLISP, could prove an ideal language.

REFERENCES

1. J. McLean, "A Formal Method for the Abstract Specification of Software," *J. ACM* **31**(3), 600-627 (1984).

2. S. Johnson, "YACC — Yet Another Compiler-Compiler," in *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution* (University of California, Berkeley, August 1983).

3. R.S. Boyer and S. Moore, *A Computational Logic* (Academic Press, New York, 1979).

4. J. Allen, *Anatomy of LISP* (McGraw-Hill, New York, 1978).

5. W.E. Clocksin and C.S. Mellish, *Programming in Prolog* (Springer-Verlag, New York, 1981).

6. J.A. Robinson and E.E. Sibert, "LOGLISP: An Alternative to Prolog," in *Machine Intelligence 10*, J.E. Hayes, D. Michie, and Y-H. Pao, eds. (Ellis Horwood, London, 1982), p. 399.

7. Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG — The Deduction Tableau Programming Language," Report STAN-CS-1012, Stanford Computer Science Technical ( ).

8. R.E. Griswold and M.T. Griswold, *The Icon Programming Language* (Prentice-Hall, Englewood Cliffs, N.J. 1983).

9. R.E. Griswold, J.F. Poage, and I.P. Polonsky, *The SNOBOL 4 Programming Language*, 2nd ed. (Prentice-Hall, Englewood Cliffs, N.J. 1971).

10. J.W. Lloyd, *Foundations of Logic Programming,* (Springer-Verlag, New York, 1984).

11. R.E. Grandy, *Advanced Logic for Applications* (Reidel, Boston, 1977).